Preprint
UCRL-JC-152873

Tool Gear: Infrastructure for Parallel Tools

J. May, J. Gyllenhaal

This article was submitted to: *2003 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV USA 6/23/03 – 06/26/03*

April 17, 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Tool Gear: Infrastructure for Parallel Tools

John May John Gyllenhaal

Abstract— ***Tool Gear is a software infrastructure for developing performance analysis and other tools. Unlike existing integrated toolkits, which focus on providing a suite of capabilities, Tool Gear is designed to help tool developers create*** **new** ***tools quickly. It combines dynamic instrumentation capabilities with an efficient database and a sophisticated and extensible graphical user interface. This paper describes the design of Tool Gear and presents examples of tools that have been built with it.***

Index Terms—**Parallel programming tools, performance analysis, automatic instrumentation, tool infrastructure.**

I. Introduction

MANY tools are available to help developers of parallel programs understand the performance and correctness of their codes. Examples include systems that show users the cost of message-passing calls, the utilization of allocated memory, or the computational efficiency of specific sections of a program. Despite this variety, many more kinds of data could be collected, and new ways could be devised for presenting it.

Researchers continue to develop new ideas for tools, but turning an idea into a usable tool can be time consuming and tedious. Most tool users are not satisfied with plain-text displays or command-line interfaces. Creating sophisticated interfaces, though, can require as much effort as building the fundamental mechanisms for gathering and processing the data. Therefore, tool researchers themselves need a set of tools they can use for building new tools.

There is more to this problem than building graphical user interfaces (GUIs): good systems for creating GUIs, such as Tcl/Tk and Java, have existed for many years. General-purpose GUI builders can make GUIs much easier to produce, but they don't help implement the underlying functionality that is common to many parallel performance tools. This functionality includes launching and controlling a target program; dynamically instrumenting the program; collecting and organizing data in a database; and presenting the data in useful ways, such as associating performance information with specific lines of source code.

Individually, most of these problems have been solved before. In fact, many of them have been solved repeatedly, and that is the main motivation for the work described in this paper. Developers of parallel tools need a tool-building infrastructure that provides common tool services so they don't have to reimplement them for each new tool. Tool Gear is a collection of programs and programming interfaces that are designed to meet this need.

Wherever possible, we have used existing open-source software. Our own contribution has been to design higher-level interfaces and to implement additional functionality. Specifically, we have:

- Designed and implemented a general-purpose client-server structure for program analysis tools. This software includes a server (or "Collector") portion that controls and instruments programs, and a Client portion that stores, analyzes, and presents data.
- Developed an extensible and sophisticated user interface that displays hierarchical views of target programs, allows users to insert and remove instrumentation easily, and annotates the source code display with performance or other data.
- Designed and implemented a program control and data collection engine that tool builders can easily augment with new types instrumentation.

Together, this software offers tool builders a straightforward way to deveop tools with a variety of sophisticated features. Our goal has been to simplify the implementation of the most common features of parallel tools while offering researchers the flexibility to gather and display many kinds of data. Because Tool Gear source code is freely available, developers can adapt it to meet their own needs.

It is important to distinguish between the Tool Gear infrastructure and the tools built with it. Although we describe the individual tools to illustrate what can be done with Tool Gear, the design and development the Tool Gear infrastructure is the research focus of this paper. Throughout this paper, the terms "Tool Gear" and "infrastructure" refer to the general-purpose software we have developed, while "tool" refers to an individual tool built using Tool Gear.

II. Related Work

Because Tool Gear aims to automate common features of parallel programming tools, many Tool Gear functions have appeared in other tools. However, unlike most parallel programming tools, the focus of this work is on the infrastructure for gathering and presenting information, and not on the kind of information gathered or the specifics of the display. Therefore, this section will compare Tool Gear mainly with other *tool development* environments, and not with the many individual tools and toolkits that present specific kinds of information.

Early examples of toolkits for building parallel tools include Voyeur [1] (created in 1989), PARADISE [2] (1991), and POLKA [3] (1993). These toolkits helped automate the development of visualizations for parallel programs. They did not support interactive control of the program, and users had to compile instrumentation into the target program.

One of this paper's authors developed an extensible debugger for parallel programs called Panorama [4]. This tool interacted

John May (johnmay@llnl.gov), Lawrence Livermore National Laboratory, 7000 East Ave., L-561, Livermore, CA 94550.

John Gyllenhaal (gyllen@llnl.gov), same mailing address.

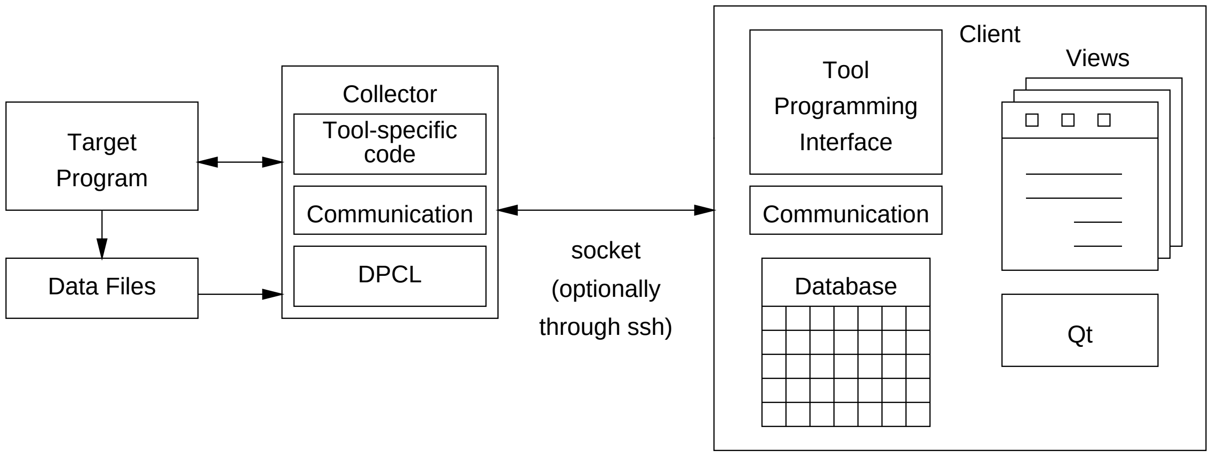


Fig. 1. Tool Gear's main components: a Collector, which includes dynamic instrumentation capability, and a Client, which stores data and manages the GUI (through Qt).

with target applications through standard command-line debuggers, and users could build Tcl/Tk-based views with the help of an interface development tool.

An important advance in programming tools was the development of the Dyninst dynamic instrumentation library [5]. Tools based on it can insert nearly any kind of instrumentation into an executable program at runtime. Eliminating the need for compile-time decisions about instrumentation gives tools great flexibility. Allowing them to insert and remove the instrumentation on the fly also helps moderate the volume of data collected. A later tool based on Dyninst is called the Dynamic Probe Class Library (DPCL) [6]. Developed at IBM, it is now open source, and Tool Gear uses it extensively.

The current systems to which it is most natural to compare Tool Gear are Paradyn [7], SvPablo [8], and TAU [9].

Paradyn presents a graphical tree-structured display of a program and allows users to choose program locations at which one or more predefined performance metrics can be computed. New metrics can be defined through a Paradyn Configuration Language.

SvPablo also uses dynamic instrumentation to gather data from hardware performance counters for specified regions of code. A source code viewer annotates source lines with performance information, and data can be stored in Pablo's extensible data format.

TAU is a suite of graphical tools that implement function profiling, interval timing, hardware counter monitoring, and related operations. TAU can use both dynamic instrumentation (through Dyninst) and compiled-in (source level) instrumentation. However, users must specify instrumentation before the target program is run; they cannot currently insert it interactively.

All these tools can gather and present performance data from parallel programs using dynamic instrumentation. All offer a variety of data views, and all are at least somewhat extensible. However, Tool Gear is not intended to compete with these tools. Instead, our goal is to help tool developers create individual new tools with minimal effort. Although developers can add new functionality to any of these other systems, the resulting tool would likely include all the current functionality as well, and this functionality might be unnecessary or confusing in the new tool. Furthermore, Tool Gear was designed from the start to be a tool infrastructure rather than a specific tool, so it offers a general-purpose foundation on which a wide range of tools can be built. We want Tool Gear to be a fast and easy way for developers to create new tools.

III. Tool Gear Components

Tool Gear has two major components: a Collector and a Client (Figure 1). The Collector controls the target program and gathers data from it. The Client manages the graphical user interface, accepts commands from the user, receives and stores data from the Collector, and presents data. The Collector and Client run as separate processes, which can run on different computers. They communicate through a Unix socket, and when the two components run on separate machines, the socket is forwarded through a Secure Shell connection.

A. The Collector

The Collector runs on same computer as the target application. It is spawned at the request of the Client, and it acts as the Client's proxy, executing commands and relaying data to it. The Collector uses DPCL to control the target application, which may be sequential or parallel. DPCL can start an application, pause and resume it, and terminate it. DPCL can also insert and remove instrumentation at runtime. Using a simple DPCL instrumentation module, we have implemented a limited form of breakpoints. (See Section V-A for details.) We call this portion of Tool Gear the "Collector" and not the "server" because it is a client to DPCL's server.

The exact details of the Collector's interaction with the Client will of course vary between tools, but in general, the Collector will first describe to the Client the capabilities it supports and the attributes (columns) of the database that the Client will manage. Then it will describe the structure of the target program, giving a list of files and functions and stating the number of processes. As the Collector receives data from the instrumentation, it will forward it to the Client, which will insert it

```
#include <dpclExt.h>
#include <sys/time.h>

static struct timeval start;
void StartTimer()
{
    gettimeofday( &start, NULL );
}

void StopTimer( AisPointer ais_send_handle )
{
    struct timeval stop;
    long sec, usec;
    double result;

    gettimeofday( &stop, NULL );
    sec = stop.tv_sec - start.tv_sec;
    usec = stop.tv_usec - start.tv_usec;
    result = sec + usec / 1e6;
    Ais_send( ais_send_handle, &result,
        sizeof(result) );
}
```

Fig. 2. A simple implementation of interval timer instrumentation. A more sophisticated version (which we have implemented) would maintain thread-specific stacks of start times so that timer calls could be nested and work correctly in multithreaded programs. The call to *Ais_send* transmits the result to the Collector, which activates a user-defined callback function to process the data.

in a database. The Collector can gather data through DPCL, or by reading files that the target program writes, or by any other means that the tool builder wishes to implement. Section IV describes how the Collector gathers data from programs.

B. The Client

The Client manages the user interface, receives and stores data, and presents graphical displays.

At the heart of the Client is a database that stores the structure of the target program, a list of actions defined by the tool that gather data or otherwise interact with the program, and the data itself. This database was originally written by one of this paper's authors for a compiler project [10]. It is well suited to Tool Gear's needs because it offers high-speed insertion and retrieval. It is not designed to manage disk-resident datasets, but we expect that limiting data to what can fit in memory will not be a major drawback for most tools. (Tools that generate very large trace files would probably do best to preprocess these files before sending the results to the Client database.)

The basic view of a target program that the Client presents to a user is a hierarchical listing of source code. This is called a Tree View (Figures 5 and 6 in Section V.) When the target program first begins running, the Tree View presents a list of the target's source files. Clicking on a file name displays the functions in that file, and clicking on a function name presents a source code listing, if one is available. If no source code is available, the Tree View displays a list of program locations that it knows about (normally, a list of function sites.)

The database is organized as a hierarchy that matches this display. Tools gather data and associate it with specific locations in the source code. The Tree View then displays this data in columns at the appropriate location. The Tree View can "roll up" data from lines into summaries for each function and file. It can also manage data separately for individual threads and processes, or summarize the data in ways specified by the tool or by the end user. Finally, the user can sort the program listing by the values in any column of the display. This could be used, for example, to find the code sections with the highest cache miss rates. (See Figure 6 for another example.)

We expect that the Tree View display will be useful for a variety of tools. However, we are also implementing graphical views that tools can use to present data in other forms. Furthermore, tool designers can implement their own custom views by programming them directly. Because data is stored centrally in the Client, and not in any one view, the Client can notify the views as new data arrives. This allows multiple views to update themselves simultaneously. Also, tools can spawn new views that show existing data in different ways.

The Client can write the contents of its database into a snapshot file at the user's request. These snapshots can be read back into a new window so the user can refer to that data as the program continues to run. A snapshot can also be used as a baseline for a display that shows the difference between each current value in the database and the corresponding value in the snapshot file.

To create these GUI displays, we use a C++ graphical interface package called Qt, developed by Trolltech. We chose Qt over other GUI tools such as Java and Tcl/Tk because it combines a rich set of features with good performance. Since we wrote Tool Gear in C++, using Qt also avoided language interoperability problems. Qt runs on a wide range of platforms, including many Unix varieties, Windows, and Macintosh OS X. It has a licensing option that permits free noncommercial use, so tool developers using Tool Gear do not need to pay a Qt license fee.

An important advantage to running the Client on a separate computer from the Collector is that it can manage the GUI locally. The graphics updates don't have to travel over a Secure Shell connection, so the GUI is very responsive, even when communicating with the Collector over low-bandwidth or encrypted connections.

In addition to the database and the viewers, the Client includes a Tool Programming Interface that helps tools define what actions users can perform on the target program. Section IV-C describes how tools use the TPI.

IV. Building Tools

Building a tool using Tool Gear consists of three tasks:

- Writing instrumentation code that will run as part of the target program.
- Writing code that tells the Collector about the instrumentation and how to forward the data to the Client.
- Using the Tool Programming Interface to define how the user can interact with the program and how the Client will display the data it receives.

The following subsections describe these steps. Our software distribution includes extensive documentation to assist developers with this process, including recipes for building tools, example code, and a complete set of Web pages that describe the programming interface and all the source code.

A. Writing Instrumentation

Tools may use Tool Gear's dynamic instrumentation capabilities to insert the instrumentation at runtime, or they may include instrumentation libraries that are compiled and linked into the target program.

To use dynamic instrumentation, the tool builder will write one or more functions (usually in C or C++) that can execute within the target program. These functions can do anything that a function written as part of the target can do, but they are compiled separately and they *do not* need to be linked into the target program. Instead, the instrumentation functions for a tool are compiled into a separate *probe module.* Figure 2 shows a simple example of instrumentation that implements an interval timer. The Tool Gear documentation describes how this would be compiled into a probe module.

To send data from the target program to the Collector, the probe module functions use DPCL's *Ais_send* function. This function transmits an arbitrary-size block of data to the Collector. The collector invokes a callback function to handle this data, as described in Section IV-B.1.

When a tool uses static instrumentation, tool developer must arrange for the Collector to receive the data. The easiest way to do this is for the instrumentation to write a file, which the Collector reads at the appropriate time. For example, the Collector can be programmed to call a function that reads a file after the target program exits. Section IV-B.2 describes this technique.

B. Modifying the Collector

The second step in building a tool is to modify Tool Gear's basic Collector program to gather and process data. How this is done will depend on whether the tool uses dynamic or static instrumentation.

B.1 Dynamic Instrumentation

When the tool uses dynamic instrumentation, the Collector's job is to cause DPCL to install specific functions from the probe module at program locations chosen by the end user. The current version of DPCL can install instrumentation only at specific locations in the target program: function entry and exit points, and just before or after any function call. These locations are called *instrumentation points,* and instrumentation that DPCL installs at one of these points is called a *point probe*. DPCL can also cause instrumentation to be executed at specified time intervals (this is called a *phase probe*) or exactly once (a *one-shot probe*.)

Tool Gear defines a set of C++ classes to encapsulate these ideas. An *action type* represents an instrumentation function, and a *point action* represents an action type that has been installed at a specific instrumentation point. The same action type can be installed at multiple instrumentation points, and a single instrumentation point can accept multiple point probes. Tool Gear also defines classes for phase probes and one-shot probes.

When defining an action type, the tool developer can specify the callback function that will be invoked whenever the corresponding instrumentation function transmits data through *Ais_send*. For point probes, the callback can determine the instrumentation point at which the function was called, so it can associate data with a particular program location.

The main steps for modifying a Collector to use the new instrumentation are these:

- Instantiate a set of action types.
- Define the data attributes in the Client database. This includes naming the columns in the database and specifying their types. Tool Gear includes functions for doing this from the Collector.
- Define the callback functions that will receive the data from the target program and forward it to the Client database.

All of this can be done in a file that is linked to the Collector, and the Collector's main function can call an initialization function in this file to set up all the actions. (We have considered designing the Collector to dynamically load the code that defines a tool's features, but we prefer to avoid the complexity and portability problems that this approach would entail.) A Collector can link in and call multiple sets of action initialization functions. This allows a tool builder to incorporate some standard action types, such as breakpoints and interval timers, along with the tool-specific action types.

The Collector already includes the ability to find the instrumentation points in a program, report them to the Client, and instantiate point actions at the Client's request. It can also instantiate point actions for a set of instrumentation points that match a pattern chosen by the user or the tool builder (such as, "entry to all functions whose names start with *MPI_*").

B.2 Static Instrumentation

Tools that rely on instrumentation that is compiled or linked directly into an application do not need to declare action types, but they do need to declare Client database attributes, and they also need to define at least one callback.

When the tool initializes itself in the Collector, it declares the attributes in the usual way, and then instead of defining action types and corresponding callbacks, it declares a single callback that the Collector will execute when the target program terminates. This routine can read a file that the program's instrumentation has written and forward the data it collects to the Client using Tool Gear's standard functions.

The instrumentation can choose a unique name for the file based on the program name and Unix process id. This information is also available to the Collector, so it can easily find the relevant file. For parallel programs, the instrumentation may write all the data to a single file, or it may write one file per process.

Tools that use dynamic instrumentation can easily determine what part of a program generated a piece of data. This allows the Collector to tell the Client database how to associate data with a program's source code. Tools that do not use dynamic instrumentation must find another way to associate data with program locations. One method is for the instrumentation to decode the target program's symbol table information so that it can look up file, function, and line number information based on the program counter. GNU libraries are available to help applications do this.

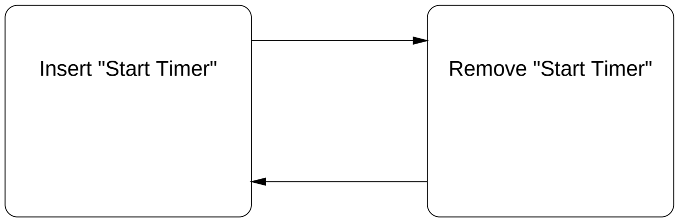


Fig. 3. Tool developers define state diagrams for each action using the Tool Programming Interface. Along with the basic states and transitions, developers specify corresponding icons and menu text.

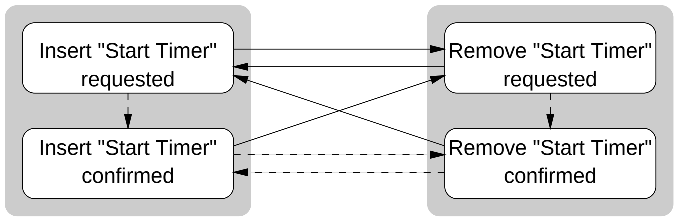

Insert "Start Timer" requested
Remove "Start Timer" requested
Insert "Start Timer" confirmed
Remove "Start Timer" confirmed

Fig. 4. Tool Gear expands the basic state diagram that the tool developer declares to include transitional states. End users can initiate only those transitions indicated by solid arrows, while the Collector can normally initiate only the transitions indicated by dashed lines. In error conditions, the Client or Collector can force the system into any state.

C. *Creating the User Interface*

The final step in defining a tool is to specify how the user will request actions and how data will be displayed.

C.1 Requesting Actions

Tool builders specify how to request actions through the Tool Programming Interface. Our model for user interactions with target programs is that each tool defines one or more actions that the user can request, and each request is associated with a location in the target program. (The “program locations” in the Client typically represent DPCL instrumentation points. We use separate terms to distinguish the objects that DPCL defines and manipulates on the target computer from those that the Client uses to represent them.) Examples of actions include inserting (or removing) some instrumentation or setting a breakpoint.

When a tool runs, tool-specific software makes calls to the Tool Programming Interface to define a set of actions, along with associated icons, menu entries, and help text. The tool also defines how actions relate to each other by defining a state transition graph (see Figure 3.) The tool uses this graph for all the program locations (or a subset of them, depending on the tool), but the Client maintains separate states for each location. The current state for a program location determines what icon is displayed, what menu choices are available to the user, and what help text is displayed when the cursor passes over an icon. The tool also defines a default initial state.

Tool Gear can automatically expand the graph that the tool defines to differentiate between *requested* actions and *confirmed* actions. It also defines additional transitions between the new states (Figure 4.) Expanding the state diagram allows the display to show different icons for requested and confirmed actions, so the user can see when a pending action has been completed. As Figure 4 shows, users can only initiate requests, while the Collector normally only confirms them. Our model for using these state transition graphs to control the interactions between the user and the collector is described more fully in a Tool Gear technical report [11].

A tool may support several independent groups of actions (such as inserting/removing instrumentation and setting/deleting breakpoints). The state diagrams that the tool defines for these groups of actions need not be connected. Thus, a program location can be in several independent states at the same time, reflecting the status of independent actions. The GUI automatically displays icons for independent states next to each other and builds the menus appropriately. For example, if the “Start Timer” action had been inserted at a particular location, clicking on that location might bring up a menu with the options of removing the “Start Timer” action, requesting a “Stop Timer” action, and requesting a breakpoint.

C.2 Using the Database

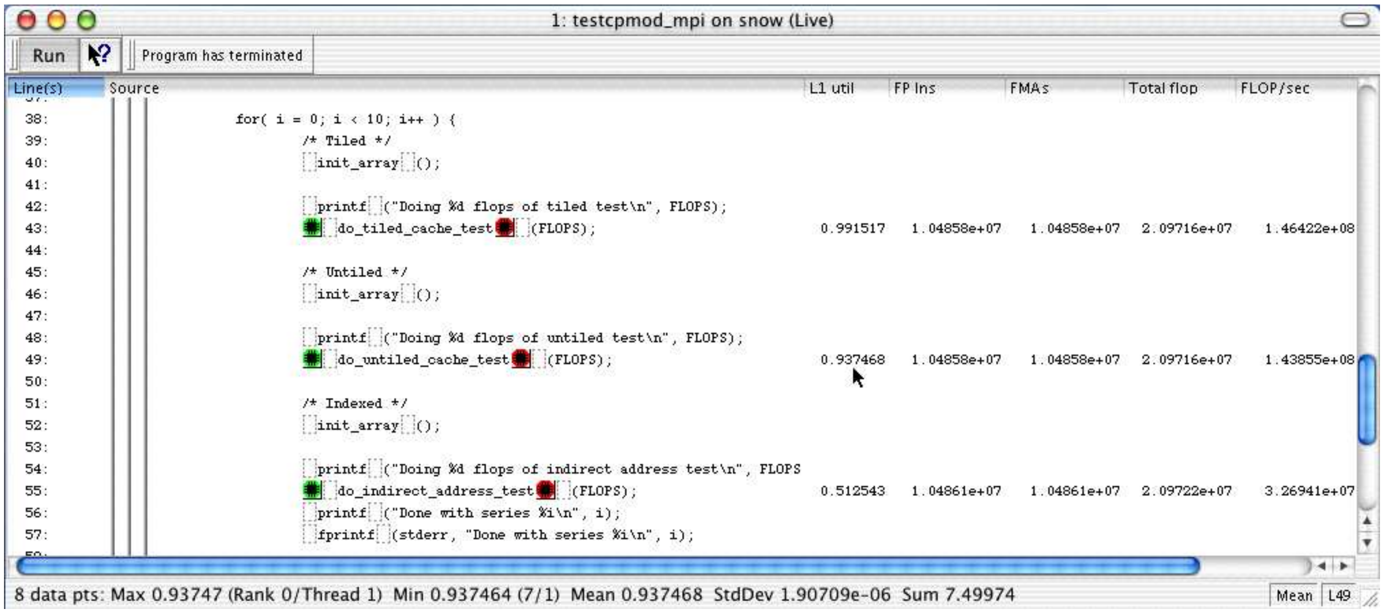
The second task in defining a user interface is determining how data will be stored and displayed. When the Collector starts, tool-specific initialization code sends requests to the Client to define one or more columns in the database, as mentioned earlier. The Client then automatically creates and labels these columns in the Tree View. Later, when the Collector receives data from the target program, it forwards it to the Client with information describing the row and column where it should be stored in the database. New values for a given cell can either replace existing ones or be added to them.

For parallel programs, the database stores data for a given row and column separately for each thread or process. The Tree View display presents a single value for a cell, which can be (at the user’s choice) the sum, mean, minimum, maximum, standard deviation, or count of the values for that row and column. The tool can specify which of these summary values is displayed by default for each column. Placing the cursor over a value shows all the summary data in a line at the bottom of the window.

The Tree View further summarizes data in each column by displaying any one of the statistics mentioned above for the function, file, and program. This could show, for example, the total time for all functions that were instrumented in a program. Again, the user can select which type of summary to display. The Client computes all these values on the fly as data arrives in the database, so there is no delay when the user chooses different values to display.

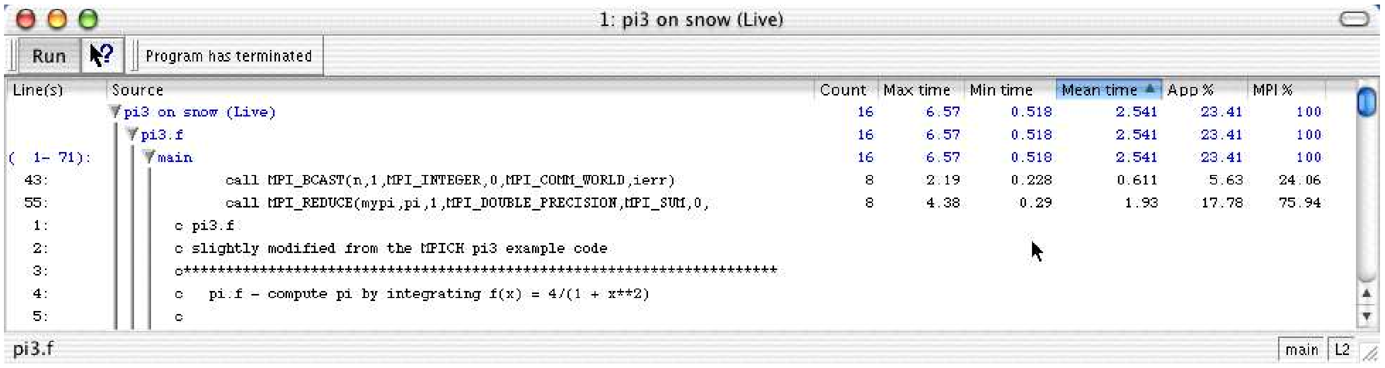
V. Tools Built with Tool Gear

We have built two prototype tools using the Tool Gear infrastructure. The first, called TGmpx, displays cache utilization and FLOP data for selected sections of code (Figure 5.) It uses dynamic instrumentation. The second, called TGmpip, shows the cost of certain MPI calls. It uses instrumentation linked in through MPI’s standard profiling interface.



```
1: testcpmod_mpi on snow (Live)
Run   Program has terminated
Line(s) Source                                                              L1 util   FP Ins        FMAs          Total flop    FLOP/sec
38:        for( i = 0; i < 10; i++ ) {
39:            /* Tiled */
40:            init_array();
41:
42:            printf("Doing %d flops of tiled test\n", FLOPS);
43:            do_tiled_cache_test(FLOPS);                               0.991517  1.04858e+07   1.04858e+07   2.09716e+07   1.46422e+08
44:
45:            /* Untiled */
46:            init_array();
47:
48:            printf("Doing %d flops of untiled test\n", FLOPS);
49:            do_untiled_cache_test(FLOPS);                             0.937468  1.04858e+07   1.04858e+07   2.09716e+07   1.43855e+08
50:
51:            /* Indexed */
52:            init_array();
53:
54:            printf("Doing %d flops of indirect address test\n", FLOPS
55:            do_indirect_address_test(FLOPS);                          0.512543  1.04861e+07   1.04861e+07   2.09722e+07   3.26941e+07
56:            printf("Done with series %i\n", i);
57:            fprintf(stderr, "Done with series %i\n", i);
8 data pts: Max 0.93747 (Rank 0/Thread 1)  Min 0.937464 (7/1)  Mean 0.937468  StdDev 1.90709e-06  Sum 7.49974      Mean  L49
```

Fig. 5. TGmpx shows cache utilization and FLOP rates for three function calls in a parallel program. The calls were instrumented at runtime, and resulting data is shown on the corresponding source lines. A statistical summary of the cell under the cursor appears at the bottom of the display. This screen shot, taken from a Macintosh OS X computer, does not show the main menus. These appear at the top of the screen on a Mac and at the top of the window on other platforms.



```
1: pi3 on snow (Live)
Run   Program has terminated
Line(s)     Source                                                              Count  Max time  Min time  Mean time  App %   MPI %
            pi3 on snow (Live)                                                  16     6.57      0.518     2.541      23.41   100
              pi3.f                                                             16     6.57      0.518     2.541      23.41   100
( 1- 71):       main                                                            16     6.57      0.518     2.541      23.41   100
  43:              call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)        8      2.19      0.228     0.611      5.63    24.06
  55:              call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,    8      4.38      0.29      1.93       17.78   75.94
   1:         c pi3.f
   2:         c slightly modified from the MPICH pi3 example code
   3:         c**************************************************************************
   4:         c    pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
   5:         c
pi3.f                                                                                                                  main  L2
```

Fig. 6. TGmpip shows the cost of MPI communications. The user has sorted the display by the Mean time column (highlighted), so the source lines with data in that column have been pulled out of the rest of the code and arranged in increasing order. The top three lines show the sum of the data in each column for the program, fi le, and function. The program has only one function, so all three are the same here. Unlike Figure 5, the cursor here is pointing at an empty cell, so the the bottom line shows the fi le, function, and line where the cursor is pointing.

A. TGmpx

TGmpx gathers hardware performance counter data using the the MPX library [12]. On systems whose counter architecture would otherwise prevent concurrent counting of certain combinations of events, MPX uses time sharing to sample counters and extrapolate results.

For TGmpx, we defined functions to gather specific combinations of data, and we turned these into DPCL probe modules. We defined two action types: one to start the measurement, and one to stop it and report the results. Starting the counters requires no callback function in the Collector. Stopping the counters invokes a callback in the Collector that simply forwards the data values (cache hit rate, FLOP count, and FLOP rate) back to the Client, along with a tag that identifies the instrumentation point where the callback was invoked and a process/thread identifier.

This tool also has an action type to implement simple breakpoints. When the user sets a breakpoint, DPCL installs a function in the target that sends an empty message back to the Collector and then puts the process to sleep for a short time. When the callback in the Collector receives this message, it tells DPCL to pause the program's execution. The sleep call in the instrumentation function gives the Collector time to receive the message and call DPCL before the program continues to the next instruction.

Figure 5 shows data for an eight-process parallel program. The tool also works for sequential programs and multithreaded programs.

All of the display capability is built into the Tool Gear infrastructure. This includes not only the features described so far but also the ability to search in the source code, customizable tool tips that present help text for various elements of the display, and a programmable "About..." box. The parts of the user interface that are unique to TGmpx are the icons, the menu text (not shown here), and the column labels.

B. TGmpip

TGmpip displays data generated by mpiP [13], a tool that instruments MPI calls and writes out a summary file describing how much time certain calls took to complete. It helps users find the communication calls in their codes that are taking a disproportionate amount of time.

This tool relies on linked-in instrumentation, so it does not need any action type declarations. Instead, we simply use DPCL to run the program. When the target finishes execution, the Collector invokes a callback to find and read the output file (whose name is based on the executable name, the number of processes, and the process id—all information that is available within the Collector). This file lists call sites for each instrumented MPI function, and for each function, there is a per-process listing of the number of times the function was called; the minimum, maximum, and mean execution time; and the percentage of time this MPI call accounts for in the total communication time and total application running time.

The tool-specific callback function culls this information from the file and transmits the per-call information to the Client for display (Figure 6).

Because this tool outputs no data until the program has finished running, there is no need for breakpoints or dynamic instrumentation. Therefore, we initially considered implementing a Collector for this tool that didn't use DPCL. However, we quickly realized that the new Collector would still need to start the parallel program, detect when it had finished executing, identify the processes, and serve source code to the Client. All of this is certainly feasible, but we decided it would be simpler to take advantage of the functionality that already exists in DPCL and not write a new version of the Collector that didn't use it. As a result, the tool-independent part of the TGmpip Collector is the same as what TGmpx uses. The disadvantage of this approach that it limits the Collector to running on systems where DPCL is available, currently IBM and Linux.

C. Tool Development Time

One of our main goals for Tool Gear is to simplify the development process so that ideas can be transformed rapidly into working, usable tools. Although the ease of implementing a tool with Tool Gear is difficult to quantify, the time needed to build a tool is a reasonable measure of our success.

Because TGmpx was developed concurrently with Tool Gear itself, it is difficult to estimate how long it would have taken to build TGmpx on top of the existing Tool Gear infrastructure.

For TGmpip, we needed to extend the Tool Gear infrastructure somewhat to handle data from static instrumentation. This took about a week, starting from the existing Tool Gear infrastructure and the stand-alone text-based tool that gathers the data.

Perhaps the best indication of how long it takes to create a new tool with Tool Gear is our experience with the interval timer tool. This tool simply measures and reports the elapsed time from a Start Timer call to a Stop Timer call. Users insert these calls using the standard dynamic instrumentation model we have described. A simplified version of the source code for the instrumentation appears in Figure 2. Implementing this tool took about half a day and required no changes to the infrastructure. The process closely followed the steps we have outlined in this paper.

VI. Current Status and Future Work

Source code for Tool Gear and the sample tools TGmpx and TGmpip are currently available free of charge from `http://www.llnl.gov/CASC/tool_gear/download/tool_gear_agree.html`. DPCL currently runs only on IBM and Linux systems, and the Linux port is experimental. However, IBM has released this code as open source, and the Dyninst technology on which it is based has been ported to many platforms. The Client, on the other hand, is already quite portable. It has been tested on IBM, Solaris, Linux, and Macintosh OS X workstations.

As other developers begin to build tools with Tool Gear, we will need to refine and extend the programming interfaces and add new functionality, especially for displaying data in different ways.

With Tool Gear's flexibility, ease of use, and sophisticated interface features, we believe it has an exciting future as a foundation on which many useful new tools can be built.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. UCRL-JC-XXXXXX.

References

[1] David Socha, Mary L. Bailey, and David Notkin, "Voyeur: Graphical views of parallel programs," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, vol. 24, no. 1, pp. 206–215, January 1989.

[2] James Arthur Kohl and Thomas Casavant, "Use of PARADISE: A meta-tool for visualizing parallel systems," in *Proceedings of the Fifth International Parallel Processing Symposium*, 1991, pp. 561–567.

[3] John T. Stasko and Eileen Kraemer, "A methodology for building application-specifi c visualizations of parallel programs," *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, pp. 258–264, June 1993.

[4] John May and Francine Berman, "Retargetability and extensibility in a parallel debugger," *Journal of Parallel and Distributed Computing*, vol. 35, no. 2, pp. 142–155, June 1996.

[5] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille, "Dynamic program instrumentation for scalable peformance tools," in *Proceedings of the Scalable High Performance Computing Conference*, May 1994, pp. 841–850.

[6] Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth, "The Dynamic Probe Class Library—An infrastructure for developing instrumentation for performance tools," in *Proceedings 15th International Parallel and Distributed Processing Symposium*, April 2001.

[7] Barton P. Miller, Mark D. Callaghan, Johathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall, "The Paradyn parallel performance measurement tools," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, November 1995.

[8] Luiz DeRose and Daniel A. Reed, "SvPablo: A multi-language architecture-independent performance analysis system," in *Proceedings of the International Conference on Parallel Processing*, September 1999.

[9] Bernd Mohr, Darryl Brown, and Allen Maloney, "TAU: A portable parallel program analysis environment for pC++," in *Proceedings of CONPAR94—VAPP VI*, September 1994, pp. 29–40.

[10] John C. Gyllenhaal, W. W. Hwu, and B. Ramakrishna Rau, "HMDES version 2 specifi cation," Tech. Rep. IMPACT-96-03, University of Illinois, Urbana, Illinois, 1996, `http://www.crhc.uiuc.edu/~gyllen/`.

[11] John May and John Gyllenhaal, "Tool Gear: Infrastructure for building parallel programming tools," Tech. Rep. UCRL-JC-147901, Lawrence Livermore National Laboratory, Livermore, California, 2002.

[12] John M. May, “MPX: Software for multiplexing hardware performance counters in multithreaded programs,” in *Proceedings 15th International Parallel and Distributed Processing Symposium*, April 2001.

[13] Jeffrey S. Vetter and Michael O. McCracken, “Statistical scalability analysis of communication operations in distributed applications,” in *Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 2001.